Processing

Creative Coding and Computational Art

Ira Greenberg

friendsof
DESIGNER TO DESIGNER™
an Apress® company

Processing: Creative Coding and Computational Art

ISBN-13: 978-1-59059-617-3

ISBN-10: 1-59059-617-X

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Credits

Lead Editor
Chris Mills

Technical Editor
Charles E. Brown

Technical Reviewers
Carole Katz, Mark Napier

Editorial Board
Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Jeff Pepper, Dominic Shakeshaft, Matt Wade

Project Manager
Sofia Marchant

Copy Edit Manager
Nicole Flores

Copy Editor
Damon Larson

Assistant Production Director
Kari Brooks-Copony

Production Editor
Ellie Fountain

Compositor
Dina Quan

Artist
Milne Design Services, LLC

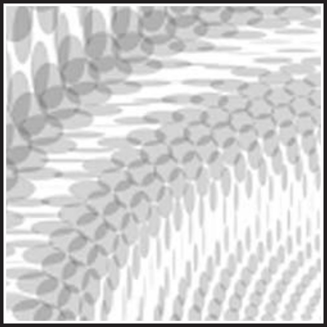
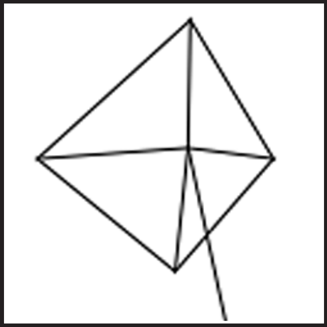
Proofreaders
Linda Seifert and Nancy Sixsmith

Indexer
John Collin

Interior and Cover Designer
Kurt Krames

Manufacturing Director
Tom Debolski

C INTEGRATING PROCESSING WITHIN JAVA

This final appendix includes information on integrating the Processing core graphics library within Java projects created independently of the Processing environment. In addition, some Processing and creative coding resources and references are included.

This appendix assumes a basic understanding of Java, including how to compile and run a Java program. I will review these procedures, but in a cursory fashion. To learn more about Java, I recommend reviewing Sun's online Java tutorials at `http://java.sun.com/docs/books/tutorial/`*, or getting a good book on Java.*

Processing outside of Processing

Because Processing is ultimately Java, it is possible to incorporate the Processing language commands into Java projects built completely outside of the Processing environment. Processing is composed of two distinct components: the Processing IDE (integrated development environment) and the Processing API (application programming interface). The IDE is just a development environment (albeit a very user-friendly one) that allows you to write, run, and view your Processing programs. The API (the actual commands making up the language) is an independent library of code accessible from within the IDE and structured as a Java package, called `processing.core`.

Packages in Java are simply directory structures (including subdirectories) containing Java classes and, in the words of Wikipedia, used to "organize classes belonging to the same category or providing similar functionality." (See `http://en.wikipedia.org/wiki/Java_package`*.)*

Packages also keep the naming of Java classes (the namespace) organized, preventing naming conflicts from occurring. Java is a very expandable language, in that anyone can create classes and packages, which can then easily be used by other coders. One of the potential risks of a structure like this is that it's possible (actually very likely) for developers to specify the same names for different classes—referred to as a name collision. This potential, left unchecked, can lead to annoying and hard-to-track-down bugs. Through the use of packages, class names are evaluated to the package name they reside within. For example, if I create a `Ball` class in a package named shapes, the full class name will resolve to `shapes.Ball`. Astute readers may then ask what happens if there is another package named shapes that also contains a `Ball` class? This too could be a likely scenario. To avoid package/class name conflicts, a unique name is often specified for the base package structure. For example, if I was going to create a library of classes, either for my own later reuse or eventual distribution, I'd use my domain name for the name of the base package, which is unique (among domain names). Keeping with the `shapes.Ball` class, here's what the complete directory structure could look like (note that subdirectories follow the forward slash character [/]):

```
com/greenberg/shapes/Ball
```

It would of course also be possible to use greater specificity, adding additional subdirectories, in defining a package structure. Here's an example:

```
com/greenberg/geometry/shapes3D/Ball
```

As you might imagine, setting up a large library of interrelated classes and packages can be a daunting organizational challenge (not to mention a technical one).

Returning to Processing, the Processing software is composed of the following 14 Java packages:

```
antlr
antlr.java
processing.app
processing.app.preproc
processing.app.syntax
processing.candy
processing.core
processing.dxf
processing.net
processing.opengl
processing.pdf
processing.serial
processing.video
processing.xml
```

These packages contain classes that, among other things, create the Processing development environment, specify the language commands, and add specialized library features. You can learn more about these packages, including viewing the classes they contain, at `http://dev.processing.org/reference/everything/`. For our purpose—learning how to utilize the Processing API (the language commands) outside of the Processing environment—we're interested in the `processing.core` package.

`processing.core` includes 12 classes that control most of the Processing language API, as well as the graphics context—the structure needed to actually draw stuff to the screen in Java. To incorporate Processing's graphic functions within your Java projects, you need to explicitly import the `processing.core` package. You can view the 12 classes in the `processing.core` package at `http://dev.processing.org/reference/everything/javadoc/processing/core/package-summary.html`.

Before describing how to actually integrate the `processing.core` package into a Java application, I want to provide a simple example demonstrating how to code, compile, and run a simple Java program. To keep things really simple, this initial program will only output some text. To run these next examples, you'll need to have the Java Standard Edition (SE) Development Kit (JDK) installed. I won't be covering how to set up Java on your respective platform, but you can find system-specific info here:

- **Mac OS X users**: `www.apple.com/macosx/features/java/`
- **Windows and Linux users**: `http://java.sun.com/javase/downloads/index.jsp`

You'll be running the Java examples from the command line, so you'll also need to know how to access a shell or terminal application on your system: Here are some additional links to help with that:

- **OS X**: `www.apple.com/macosx/features/unix/` and `www.simplehelp.net/2006/10/05/an-introduction-to-the-os-x-terminal/`
- **Windows**: `http://java.sun.com/docs/books/tutorial/getStarted/cupojava/win32.html`
- **Linux**: `http://java.sun.com/docs/books/tutorial/getStarted/cupojava/unix.html`

Finally, you'll need a simple text editor to write the Java programs. Notepad on Windows and TextEdit on OS X will work. However, there are also many other free (and better) text editors you can download off the Web. Here are links to some text editors:

- **OS X**: `www.barebones.com/products/textwrangler/index.shtml`
- **Windows**: `www.crimsoneditor.com/`
- **Linux**: `http://bluefish.openoffice.nl/index.html`

In addition to a plain vanilla text editor (or the Processing environment), there are other IDEs you can use to write Java programs. One of the more popular of these is Eclipse (`www.eclipse.org/`). Eclipse is an industrial strength open source development environment. Programs like Eclipse, once mastered, really speed up the software development process; that being said, I'll only be showing you how to run the examples from the command line. I have nothing against IDEs (some of my best friends use them), but it will be much easier to understand precisely how the Processing/Java integration works by doing it "by hand," so to speak.

Creating your first Java program

Launch your trusty text editor and enter the following code into it:

```
public class MyFirstJavaProgram {

  // constructor
  public MyFirstJavaProgram(){
    System.out.println("Waaaaaz up World!");
  }

  // main method
  public static void main(String[] args){
    new MyFirstJavaProgram();
  }
}
```

Then save the file, naming it exactly the same name as the class. (Remember that Processing and Java are case sensitive.) You also need to add `.java` to the end of the name. I named the file `MyFirstJavaProgram.java`.

If you've read the entire book in order—learning about OOP in Chapter 8 and then how to work in Java mode in Chapter 14—most of the code in `MyFirstJavaProgram.java` probably looks familiar. However, if you're like me, and think that *linearity* is a bad word, this code might need some clarification.

Notice the `public` keyword preceding the class declaration, class constructor, and `main()` method. In Processing, when you create a class (working in continuous mode), you don't add the `public` keyword (which is referred to as an access modifier). However, when your Processing code is converted to standard Java, these modifiers are automatically put in for you. You can confirm this by exporting one of your Processing sketches and then opening up the `.java` file created.

Next, I'll describe the different parts of the Java code. First is the constructor:

```
// constructor
  public MyFirstJavaProgram(){
    System.out.println("Waaaaaz up World!");
  }
```

The constructor is the method that's called when an object of the class is created (instantiated), in this case through the expression `new MyFirstJavaProgram()`. I discussed constructors in numerous places throughout the book, so I'll assume you get the concept; if not, please refer to Chapter 8.

What's probably new to many of you is the `main()` method:

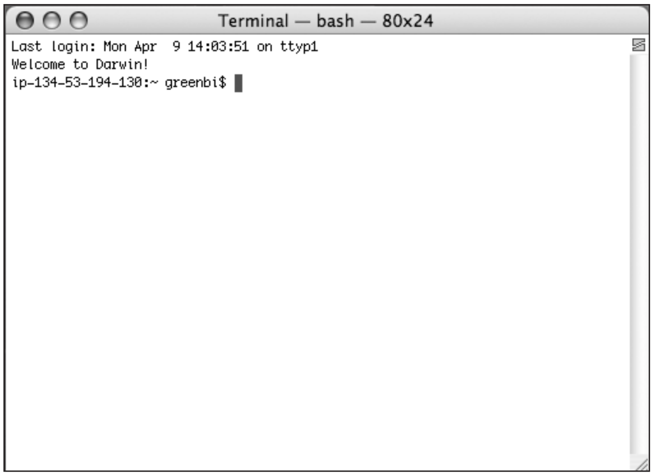
```
// main method
  public static void main(String[] args){
    new MyFirstJavaProgram();
  }
```

A `main()` method is required to execute a Java program. In fact, this is where program execution begins. The `static` keyword specifies that the method is a class method (as opposed to an instance method). Static methods are called using the class name, preceding the method name, via dot syntax (e.g., `MyClass.statusMethod()`). You do not instantiate an object to call a static method. The `main()` method is unique in that you don't explicitly call it. Rather, when you execute the program, which you'll learn how to do in a moment, the `main()` method is invoked automatically. Lastly, notice that from within the `main()` method, I instantiate an object of the class (`new MyFirstJavaProgram();`), which you'll remember calls the class constructor mentioned earlier. OK, so now let's run our exciting first Java program.

Running your first Java program

Well, there is actually one more step you need to do before you can run your program. The Java code you write needs to be compiled into what's referred to as bytecode. **Bytecode** is the code the Java interpreter reads as it executes your program. Compiling your `.java` files will create `.class` files of the same exact name. For example, compiling the file `Program1.java` will generate `Program1.class`; and again, it's the compiled `.class` files that are run.

To compile and run the program, you'll need to open your shell or terminal program. Figure C-1 shows a screenshot of my terminal program in OS X.



```
Last login: Mon Apr  9 14:03:51 on ttyp1
Welcome to Darwin!
ip-134-53-194-130:~ greenbi$
```

Figure C-1. OS X terminal application screenshot

You'll need to navigate to the directory where the `MyFirstJavaProgram.java` file is saved. I put the file in a directory named appc on my desktop.

In the terminal program in OS X, I entered the following UNIX command to navigate within the appc directory (obviously, you'll need to enter the commands specific to your system and terminal application and relative to where you saved the `MyFirstJavaProgram.java` file):

```
cd desktop/appc
```

To then confirm that the file is in the directory, I entered the following command:

```
ls
```

The `dir` *command should work accordingly on Windows.*

This then outputs the following:

```
MyFirstJavaProgram.java.
```

Now it's time to compile the program. Assuming that you have successfully navigated, via your command-line tool, to the directory in which `MyFirstJavaProgram.java` is stored, and that the Java JDK is successfully installed, you should type the following:

```
javac MyFirstJavaProgram.java
```

Javac is a compiler application that comes with the Java JDK and compiles your `.java` files to `.class` files (bytecode). After entering the `javac MyFirstJavaProgram.java` command, look again inside the directory where `MyFirstJavaProgram.java` is saved. You should now see two files listed:

```
MyFirstJavaProgram.class
MyFirstJavaProgram.java
```

Congratulations! You just compiled your first Java program. The next step is easy—running your program. Simply enter the following command:

```
java MyFirstJavaProgram
```

Hopefully your program outputs the following (within your terminal program):

```
Waaaaaz up World!
```

Pretty easy, right? Adding the `processing.core` package into the mix involves a few more steps, but not many.

Next, you'll add a second really, really simple class, which you'll reference from the first class. Create a new file in your text editor and enter the following code:

```
public class MySecondJavaProgram {

  // constructor
  public MySecondJavaProgram(String name){
    System.out.println("Hello there " + name);
  }
}
```

Save this file as `MySecondJavaProgram.java` in the same directory as `MyFirstJavaProgram.java`. You'll also need to make a simple change to `MyFirstJavaProgram.java`. Following is the updated code. Notice that the only change (displayed in bold) is to the constructor.

```
public class MyFirstJavaProgram {

  // constructor
  public MyFirstJavaProgram(){
   new MySecondJavaProgram("Creative Coder");
  }
```

```
    // main method
    public static void main(String[] args){
      new MyFirstJavaProgram();
    }
  }
```

Returning to your terminal program, recompile `MyFirstJavaProgram.java`:

```
javac MyFirstJavaProgram.java
```

If you look at your directory again, where the `.java` files are saved, you should now see four files in the directory:

```
MyFirstJavaProgram.class
MyFirstJavaProgram.java
MySecondJavaProgram.class
MySecondJavaProgram.java
```

Please note that you don't need to explicitly compile the new `MySecondJavaProgram.java` *file, as the compiler will automatically compile any other referenced classes. Also, when compiling, files will be overwritten by the same named files without warning.*

Now try running `MyFirstJavaProgram` again:

```
java MyFirstJavaProgram
```

You should see the following output:

```
Hello there Creative Coder
```

Java packages

The last thing you need to understand before plugging in the `processing.core` package is how to both specify and then communicate with a Java package.

Create a subdirectory named `pkg1` in the directory where the `MyFirstJavaProgram.java` and `MySecondJavaProgram.java` files reside. Then move the file `MySecondJavaProgram.java` into the `pkg1` subdirectory. (If you'd like to, you can also delete the two class files, `MyFirstJavaProgram.class` and `MySecondJavaProgram.class`, from the main directory. However, it is not necessary to do so.)

Next, you need to update the Java file `MySecondJavaProgram.java`, specifying the `pkg1` package in which it now resides; this just requires a package declaration statement to be added to the top of the class (shown in bold in the following code):

```
package pkg1;

public class MySecondJavaProgram {

  // constructor
  public MySecondJavaProgram(String name){
    System.out.println("Hello there " + name);
  }
}
```

Again, a package is nothing more than a directory. However, once packages are introduced into Java, compiling and running your Java programs gets more complicated (an issue that unfortunately needs to be addressed if you eventually want to incorporate processing.core in Java projects).

Returning to your terminal program, if you try to recompile MyFirstJavaProgram.java now, you'll get an error such as the following:

```
MyFirstJavaProgram.java:7: cannot find symbol
symbol : class MySecondJavaProgram
location: class MyFirstJavaProgram
   new MySecondJavaProgram("Creative Coder");
       ^
1 error
```

What's happening is that the MyFirstJavaProgram class can't find the MySecondJavaProgram class, since it's been put into the pkg1 subdirectory. Your initial thought might just be to update the path to MySecondJavaProgram.java from MyFirstJavaProgram. Let's try this.

In MyFirstJavaProgram.java, change the line

```
new MySecondJavaProgram("Creative Coder");
```

to

```
new pkg1.MySecondJavaProgram("Creative Coder");
```

and try recompiling. Sure enough, this works! The compiler can now find the MySecondJavaProgram class from within the MyFirstJavaProgram class. Although it's possible to use full *package.class* addressing as you just did, it's not really practical, as it would require a lot more code to be written, especially once you begin using multiple packages in your programs. Java includes the import keyword, which allows you to import both individual classes and groups of classes, avoiding the need to include the explicit package directory structure in each class reference.

Change the line with the explicit package reference that now reads

```
new pkg1.MySecondJavaProgram("Creative Coder");
```

back to

```
new MySecondJavaProgram("Creative Coder");
```

Then add the statement

```
import pkg1.MySecondJavaProgram;
```

to the very top of the `MyFirstJavaProgram` class (above the class declaration statement).

This file should compile successfully, since the `MySecondJavaProgram` class can again be found from within `MyFirstJavaProgram`.

Please note that package names begin with a lowercase letter, while class names begin with a capital letter.

There is one other variation that can be made to the import statement to make it even more efficient. Imagine if you created a program that relied on 20 classes defined in another package. One option would be to add the package name explicitly in front of each class reference from the package throughout your program, as we did earlier (`new pkg1.MySecondJavaProgram("Creative Coder");`). Or, you could add an import statement for each class in the package (e.g., `import pkg1.MySecondJavaProgram;`). However, these strategies would still require lots of separate long class reference or import statements.

A better solution is to import all the classes in a package in one fell swoop, which can be accomplished using the syntax `import packageName.*;`. For example, to import all the classes in a package named `geometry`, you'd write `import geometry.*;`. If you wanted to use all the classes in a subdirectory within `geometry` named `shapes2D`, you could use the import statement `import geometry.shapes2D.*;`.

core.jar

Let's now try importing the `processing.core` package into a Java program. The first step is to copy the `processing.core` package into the directory where your Java class will reside. The `processing.core` package can be found within the `lib` directory, within your Processing application directory. The package is called `core.jar`. Copy this file (being sure to not remove it from the `lib` directory) into the directory in which you've been running the Java examples.

Perhaps you're wondering why the `processing.core` package is called `core.jar` and appears to be a single file rather than a directory. The `.jar` suffix specifies a compressed, bundled file, which essentially adheres to the popular ZIP file format. The JAR format includes some other geeky stuff as well, beyond this discussion. If you're interested in learning more about the JAR format, check out `http://java.sun.com/j2se/1.4.2/docs/guide/jar/jar.html`. The JAR format is mostly used to bundle together classes and necessary resources, such as images and sounds.

The core.jar file contains numerous Java classes, as I mentioned earlier, that form most of the Processing language API. It's also possible to look inside the core.jar file to see these 12 classes, as well as how the processing.core package (within core.jar) is organized.

When the JDK is installed, a number of tools and applications come with it, such as the complier javac and a tool called jar. The jar tool allows you to create and edit JAR files as well as view and extract their contents. Lets take a look inside core.jar.

Using your command-line tool, navigate to the directory where you copied the core.jar file. Once there, type the following command:

```
jar tf core.jar
```

You should then see the following output:

```
processing/
processing/core/
processing/core/PApplet$1.class
processing/core/PApplet$2.class
processing/core/PApplet$3.class
processing/core/PApplet$4.class
processing/core/PApplet$5.class
processing/core/PApplet$6.class
processing/core/PApplet$7.class
processing/core/PApplet$8.class
processing/core/PApplet$RegisteredMethods.class
processing/core/PApplet$Worker$1.class
processing/core/PApplet$Worker.class
processing/core/PApplet$WorkerVar.class
processing/core/PApplet.class
processing/core/PConstants.class
processing/core/PFont.class
processing/core/PGraphics.class
processing/core/PGraphics2D.class
processing/core/PGraphics3D.class
processing/core/PGraphicsJava2D$ImageCache.class
processing/core/PGraphicsJava2D.class
processing/core/PImage.class
processing/core/PLine.class
processing/core/PMatrix.class
processing/core/PPolygon.class
processing/core/PShape.class
processing/core/PTriangle.class
```

Notice that the core.jar archive contains a processing directory, which contains a core directory, which then contains all the classes.

Once you know the directory structure of the package, you can begin to use it, as I did with the pkg1 package created earlier. However, working with JAR files is slightly trickier than with straight (un-jarred) packages. Next, I'll show you how to access a single value in core.jar; then I'll finally show you how to create a real Java graphics application utilizing the Processing API.

Within the processing.core directory is a PConstants class file. PConstants.class is actually an interface that contains lots of constants—thus its name. (If you don't remember what an interface is, please refer to Chapter 8.) To access a constant in Java (assuming that the constant is defined with the static keyword), you use the class name, followed by a dot and the constant name. For example, there is a constant in the PConstants interfaced named A; to refer to it, you'd write PConstants.A. Let's try this.

In MyFirstJavaProgram.java, add the following import statement to the top of the class (you can leave the import pkg1.MySecondJavaProgram; statement there as well):

```
import processing.core.*;
```

Please note that it is unnecessary to convert core.jar *back to its uncompressed package directory structure, as the compiler will automatically look within the JAR file for the necessary packages and classes.*

Next, add a second println() statement to the MyFirstJavaProgram.java constructor (displayed in bold in the following code):

```
// constructor
  public MyFirstJavaProgram(){
  new MySecondJavaProgram("Creative Coder");
  System.out.println("PConstants.A = " + PConstants.A);
  }
```

You can leave the other statement (new MySecondJavaProgram("Creative Coder")) in the constructor as well, as long as you also left the import statement (import pkg1.MySecondJavaProgram;) at the top of the class; if you didn't, make sure you comment out or remove new MySecondJavaProgram("Creative Coder") from the constructor.

In your terminal program, now try compiling MyFirstJavaProgram.java again using the following line (warning: this shouldn't work):

```
javac MyFirstJavaProgram.java
```

Trying to compile should generate the following error message:

```
MyFirstJavaProgram.java:2: package processing.core does not exist
import processing.core.*;
^
MyFirstJavaProgram.java:9: cannot find symbol
symbol : variable PConstants
location: class MyFirstJavaProgram
        System.out.println("PConstants.A = " + PConstants.A);
                                               ^
2 errors
```

I told you working with JAR files is a little more complicated than working with straight packages. The error was caused because the compiler couldn't find the `processing.core` package. You need to explicitly tell the compiler where the JAR is located, using a **classpath** argument with the `javac` command. The classpath argument tells Java where to look for class files—which is by default the current directory containing the Java file you're compiling. Here's the compile statement with the classpath argument added:

```
javac -classpath .:core.jar MyFirstJavaProgram.java
```

Since you're changing the default classpath (which includes just the current directory), you now need to explicitly include the current directory (where the Java file you're compiling is located) along with the path to `core.jar`. Use a dot to refer to the current directory.

Please note that in my example, which I coded in OS X, I used a colon to separate the current directory from `core.jar` *in the compile statement. Windows instead requires a semicolon to separate multiple paths. For more information about working with the classpath in OS X (and UNIX), please refer to* `www-128.ibm.com/developerworks/java/library/j-classpath-unix/?ca=dgr-lnxw07clspth-Unix-MacX`. *For information about the classpath in Windows, check out* `www-128.ibm.com/developerworks/java/library/j-classpath-windows/?ca=dgr-lnxw07clspth-Windows`.

Now try running the updated program again, using the following command:

```
java MyFirstJavaProgram
```

The program now outputs the following:

```
Hello there Creative Coder
PConstants.A = 6
```

Note that if you removed the line `new MySecondJavaProgram("Creative Coder")` *from the* `MyFirstJavaProgram` *constructor, you'll only get one line of output.*

I realize that the output is not terribly exciting, but the line `PConstants.A = 6` tells you that you are now indeed accessing the `processing.core` package independently of the Processing IDE (which will now allow you to do some exciting things).

Putting it all together

Two of the challenges of graphics programming in Java are setting up the drawing component (the actual window that you draw to) and correctly communicating with the graphics context of this component to control when something needs to be drawn to the screen. Working in Processing, you can pretty much take these sorts of processes for granted, since it's mostly handled for us behind the scenes. If you've ever worked with HTML tables, you know how difficult (and really annoying) it can be sometimes just to code the simplest layouts. Coding graphics in Java can be equally frustrating. However, by using the processing.core package, you can greatly simplify the graphics coding process while retaining the benefit of working in pure Java.

In the next example, I'll create two simple classes: Controller.java and EmbeddedP55.java. I'll also set up a package structure based on my domain name (iragreenberg.com). Feel free to substitute your own domain name for mine, but just make sure to be consistent in properly changing the name in any import statements as well; if you have any doubt about being able to successfully do this, just use my domain name for now.

Here's the EmbeddedP55 class:

```
package com.iragreenberg;

import processing.core.*;
import java.awt.*;

public class EmbeddedP55 extends PApplet {

  private int w, h;
  private int xc, yc;
  private float rx, ry, rw, rh;
  private float ang, radius;
  private float alph = 50.0f;
  private float spacer = 1.0f;

  public EmbeddedP55(int w, int h){
    this.w = w;
    this.h = h;
  }

  public void setup() {
    /* account for height of frame title bar,
     when sizing applet. getBounds() returns a Java
     Rectangle object that has 4 public properties:
     (x, y, width and height) */
    int titleBarHt = getBounds().y;
    size(w, h-titleBarHt);
    background(255);
    frameRate(30);
    noStroke();
    smooth();
    xc = width/2;
    yc = height/2;
  }

  public void draw(){
    if (ry < height){
      fill(50, alph);
```

```
          rx = xc + cos(ang)*radius;
          ry = yc + sin(ang)*radius;

          rw = cos(ang*spacer)*(radius/8.0f);
          rh = sin(ang*spacer)*(radius/8.0f);

          ellipse(rx, ry, rw, rh);

          ang += .075f;
          radius +=.08f;
          alph +=.001f;
          spacer +=.00001f;
        }
      }
    }
```

Enter the class into your text editor and save the class as EmbeddedP55.java, within the directory structure com/iragreenberg. In other words, create a directory named com, and then create another directory within com named iragreenberg; then save the EmbeddedP55.java file within there.

Looking at the EmbeddedP55.java class, notice that I begin the class with the following package declaration statement:

```
package com.iragreenberg;
```

Obviously, the package declaration should match where your class is indeed located. For now, it doesn't matter where the outer com directory is located, as long as you can navigate to it using your terminal program.

Following the package statement, I import the processing.core package (archived within core.jar) and the Java package java.awt. I'll discuss Java's awt package shortly, when we look next at the Controller.java class. When you download the Java JDK, an enormous library of code comes with it, including the java.awt package.

Next, I declare the EmbeddedP55 class extending Processing's PApplet class. Remember, through this inherited relationship, objects created from the EmbeddedP55 class will also be of the second type, PApplet, and have access to the (non-private) members (properties and methods) of this class. To learn more about inheritance, refer to Chapter 8, or check out http://java.sun.com/docs/books/tutorial/java/concepts/inheritance.html.

The PApplet class, as stated in the Processing reference, is the "base class for all sketches that use processing.core." This class contains methods with the same names as the function calls in the Processing API. By extending PApplet, you'll be able to refer directly to Processing's function calls from within the EmbeddedP55 class, since you refer to the methods from within the same class (or a subclass) directly—without the need to first instantiate an object.

The rest of the class is pretty much straightforward Processing code, which I believe I've explained enough throughout the rest of the book. There are three finer points, though, that might need some clarification. The first is the use of the public and private access modifiers.

In general, as I discussed in Chapter 14, methods, constructors, and the classes themselves should be declared public, while properties should be declared private. In truth, you could leave the private modifiers off, and the program would still run (the public modifiers on the other hand are required). But the convention in Java, again, is to make your properties private and then create get() and set() methods to change and access the values of the properties. I chose to not include any get() and set() methods in the example for the sake of brevity. Normally, I would include them, especially if I were creating a real reusable class.

The second point to clarify is the use of f after the decimal values in the class (e.g., float alph = **50.0f;**). By default, Processing uses float values. The float primitive type in Java is 32 bits long, while the double primitive type is 64 bits. Processing was designed based on the trade-off that a slight loss in accuracy (from 64 bits to 32 bits) was worth it for increased performance. By default in Java, all decimal values are of primitive type double. However, Processing functions can only handle float arguments. By using f as post notation after a decimal value, you can explicitly cast double values to float values, as I did throughout the class.

The last fine point is in the following lines within the EmbeddedP55 setup() method:

```
int titleBarHt = getBounds().y;
size(w, h-titleBarHt);
```

Since the PApplet component will reside within a Frame window (which includes a title bar), I wanted to account for the title bar height to allow any output to be properly centered in the PApplet component. I also wanted to be able to keep using Processing's handy width and height properties. From within the Controller class (which I'll discuss next), I passed in width and height values for the PApplet component to the EmbeddedP55 constructor. However, these values referred to the complete Frame window, including the top menu bar. By using the getBounds().y method (of Java's Component class), I was able to get the specific y position, relative to the Frame window, at which the PApplet component was placed; thus, I knew this value represented the height of the menu bar. I then sized the PApplet component to the size of the outer frame minus the height of the menu bar. I realize this is a pretty subtle point, but also one of those problems that can annoy you to no end until you solve it—which hopefully I've (more or less) done for you.

Here's the Controller class:

```
import com.iragreenberg.EmbeddedP55;
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class Controller extends Frame {

  private int w = 400, h = 422;
  // constructor
  public Controller() {
    // call to superclass needs to come first in constructor
    super("Frame with Embedded PApplet Component");

    // set up frame (which will hold applet)
    setSize(w, h);
    setLayout(new FlowLayout(FlowLayout.LEFT, 0, 0));

    // Instantiate Applet object
    Applet p55 = new EmbeddedP55(w, h);

    // add Applet component to frame
    add(p55);

    // won't allow frame to be resized
    setResizable(false);

    // allow window and application to be closed
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });

    // Next comment taken directly from PApplet class:
    /* "...ensures that the animation thread is started
     and that other internal variables are properly set."*/
    p55.init();
  }

  public static void main(String[] s){
    new Controller().setVisible(true);
  }
}
```

The Controller class is sort of the conductor class, in my example. You can also think about it as the stage for the code performance (so to speak) to take place. The Controller class is not designed to be a reusable class or something that would be included in a code library, so I won't bother putting it into a custom package. (Some Java purists—who believe fervently that all classes should be explicitly put in a package—might not like that. But hey! I'm an artist.) Simply save the class (naming it Controller.java) at the same location as the com directory.

Notice the import statements at the top of the Controller class. The first one

```
import com.iragreenberg.EmbeddedP55;
```

references the EmbeddedP55 class, saved within the com.iragreenberg directory structure. Remember, by including the import statements, you can avoid having to use the full package address when referring to a class in the package. The next three statements

```
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;
```

reference classes in existing Java packages that come with the Java JDK. Notice that java.awt.*; and java.awt.event.*; seem to almost reference the same package. The difference is that the event directory is a subdirectory of awt (not a class in awt).

*Java's Abstract Windowing Toolkit (*awt*) is a large and complex code library (12 overflowing directories chock-full of tasty code) devoted to constructing user interfaces in Java (a task not for the faint of heart). You can learn more about it at* www.javaworld.com/javaworld/jw-07-1996/jw-07-awt.html.

Remember that packages begin with a lowercase letter and classes begin with an initial cap. Also, and importantly, the wildcard syntax (.*) only refers to all the classes in a directory, not all the subdirectories in a package.

Next, notice that the Controller class declaration statement includes the extends Frame keywords, which should key you in that the Frame class will be the superclass to the Controller class.

Java's Frame class is referred to in the Java API as a "top-level window" class (see http://java.sun.com/j2se/1.4.2/docs/api/java/awt/Frame.html). You'll put your PApplet object within the literal frame window. Because the Frame class organizes graphical elements, it is equipped with layout capabilities, again not dissimilar to how a table works in HTML.

In the Controller class constructor, the first call

```
super("Frame with Embedded PApplet Component");
```

is to the Frame superclass constructor, passing in a String argument ("Frame with Embedded PApplet Component") used to title the window. Remember, any calls to super() need to occur first in the subclass constructor. Next, I set the size of the external window, as well as the layout rule for the window, passing in a FlowLayout object. The FlowLayout class is another Java class, involving complexity beyond this discussion.

In fact, layout issues are not trivial in Java. If you plan on doing a lot of coding of Java UIs, I highly recommend getting a few good dedicated books on the subject (and perhaps one of those squishy balls for stress relief). The arguments I passed into the FlowLayout instantiation statement, new FlowLayout(FlowLayout.LEFT, 0, 0), will (hopefully) ensure that the PApplet object is placed flush against the top and left edges of the frame window. Here's a link to more about the art of laying stuff out in Java components: http://java.sun.com/docs/books/tutorial/uiswing/layout/using.html.

Next, I instantiated the PApplet object, passing in the frame's width and height, which you'll remember I use in the EmbeddedP55 class to size the embedded PApplet. The line

```
add(p55);
```

literally adds the EmbeddedP55 object (also of type PApplet) into the outer frame.

The line setResizable(false); simply removes the default resizing handle in the bottom-right corner of the window. You can eventually comment out this line if you'd like to be able to resize the window as the program runs.

The next block of code is necessary if you want users to be able to close the application (not just the window itself) by clicking on the window's close button:

```
// allow window and application to be closed
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });
```

The line p55.init(); gets your embedded PApplet running.

Finally, and as we looked at earlier, the class you wish to explicitly execute needs a main() method. In the Controller class's main() method, the line

```
new Controller().setVisible(true);
```

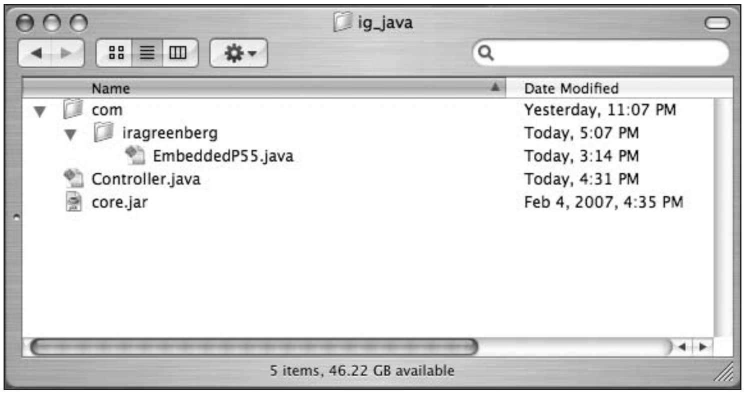
not only calls the Controller class constructor, but also makes the frame visible. You can of course put these calls on two separate lines if you prefer, like this:

```
Controller c = new Controller()
c.setVisible(true);
```

Now, let's try running the program from the command line.

Using your terminal program, navigate to where the Controller.java class and com.iragreenberg directory are saved. You'll also need to place a copy of the core.jar archive at the same location. On my system, I created a directory called ig_java, within my Documents directory, and put everything in there. (You can put the Controller.java file, core.jar archive, and com.iragreenberg directories anywhere you want on your system, as long as they stay in the same relative position to each another. You of course also need admin rights to be able to compile and execute Java from the directory they're stored in.)

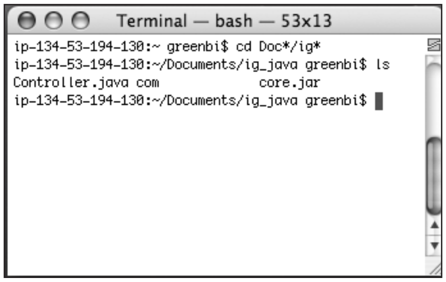
Next, I navigated, using my terminal application, to the ig_java directory. Figure C-2 shows a screenshot of the ig_java directory, and Figure C-3 shows a screenshot of my terminal program.



```
Name                         Date Modified
▼ com                        Yesterday, 11:07 PM
  ▼ iragreenberg             Today, 5:07 PM
      EmbeddedP55.java       Today, 3:14 PM
  Controller.java            Today, 4:31 PM
  core.jar                   Feb 4, 2007, 4:35 PM

5 items, 46.22 GB available
```

Figure C-2. ig_java directory



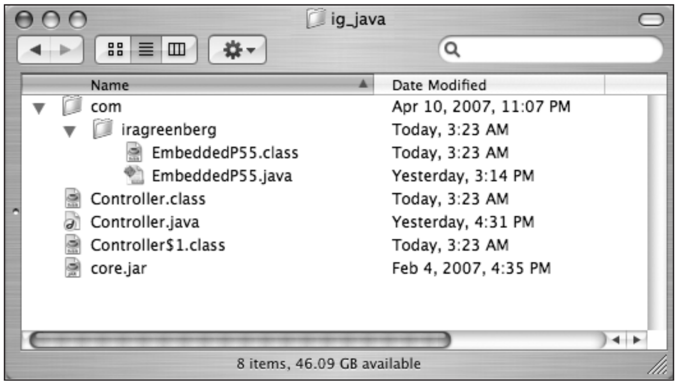
```
ip-134-53-194-130:~ greenbi$ cd Doc*/ig*
ip-134-53-194-130:~/Documents/ig_java greenbi$ ls
Controller.java com             core.jar
ip-134-53-194-130:~/Documents/ig_java greenbi$ 
```

Figure C-3. Terminal application showing the contents of ig_java directory

Within the terminal program, enter the following command to compile:

```
javac -classpath .:core.jar Controller.java
```

Successfully compiling should create a bunch of class files in your `ig_java` and `com.iragreenberg` (or whatever you named them) directories. Figure C-4 show a screenshot of my `ig_java` directory after compiling.



```
Name                         Date Modified
▼ com                        Apr 10, 2007, 11:07 PM
  ▼ iragreenberg             Today, 3:23 AM
      EmbeddedP55.class      Today, 3:23 AM
      EmbeddedP55.java       Yesterday, 3:14 PM
  Controller.class           Today, 3:23 AM
  Controller.java            Yesterday, 4:31 PM
  Controller$1.class         Today, 3:23 AM
  core.jar                   Feb 4, 2007, 4:35 PM

8 items, 46.09 GB available
```

Figure C-4. ig_java directory after compiling

Notice the class created named `Controller$1.class`*. The "$1" part of the name is not a typo, but indicates that the* `Controller.java` *class contains one anonymous inner class, which automatically got compiled. Anonymous classes by definition are classes without names. In the* `Controller` *class, the anonymous class is the argument passed in the* `addWindowListener` *call:*

```
addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });
```

Notice how the instantiation call `new WindowAdapter()` *is followed by a block of code, combining object instantiation of the class and the class declaration in the same statement. You can learn more about anonymous and inner classes at* `www.developer.com/java/other/article.php/3300881`.

To run the application, you need to include the classpath argument as well, just like you did during compiling. Next is the java command:

```
java -classpath .:core.jar Controller
```

Congratulations, you have just run your first Java graphics sketch with an embedded PApplet—woo hoo! Figure C-5 shows an output of this fabulous sketch.

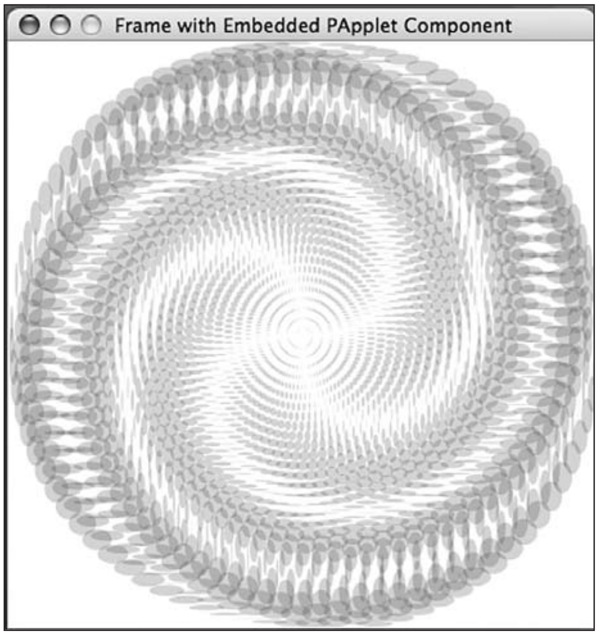


Figure C-5. First Java graphics sketch, with embedded Processing PApplet

Creating a simple GUI

The last thing I want to briefly cover is how to integrate a Processing PApplet component within a graphical user interface (GUI). On the Processing dev site, within the PApplet class, is a cautionary note about including Java GUI elements directly within Processing sketches: "You should not use AWT or Swing components inside a Processing applet. The surface is made to automatically update itself, and will cause problems with redraw of components drawn above it." (See http://dev.processing.org/reference/core/javadoc/processing/core/PApplet.html.) The note goes on to say that if you want to use AWT or Swing elements, you should incorporate the PApplet component as a separate element within a larger AWT- or Swing-based GUI.

I mentioned AWT earlier. Swing is another GUI (or widget) toolkit for Java that tried to expand and improve upon the limitations of AWT. Fundamentally, the two toolkits are different in that AWT relies on the native windowing toolkit of your respective system, while Swing components are completely written in Java. Thus, theoretically, AWT components will look like the native windows on your respective platform, while Swing should look the same on all platforms. Well, that is a nice idea, but it doesn't quite work as flawlessly as that. However, Swing does allow a lot more sophisticated types of GUI controls to be (relatively easily) added to your applications. Swing also provides for some skinning capabilities of components, allowing you to change the look and feel of the widgets. You can learn more about Swing at http://java.sun.com/docs/books/tutorial/uiswing/start/index.html.

Besides AWT and Swing, there are some other Java widget toolkits. Probably the best known is the Standard Widget Toolkit (SWT), developed by IBM as an open source alternative to Swing, which apparently IBM found too sluggish. You can learn more about SWT at www.eclipse.org/swt/.

To wrap up this appendix (as well as the entire book), I've created a playful kite toy example that I'll initially just plop in a frame, as I did in the last example. Then I'll add the kite PApplet into a slightly more sophisticated AWT application (all right, it's got one button). Finally, I'll integrate the kite into a Swing layout, providing some user control of a few of the kite's parameters.

I'll be building these applications, again, in the ig_java directory. Initially, I'll create two new classes: KiteController.java and Kite.java. I'll put the KiteController.java class within the ig_java directory and I'll put Kite.java within com.iragreenberg (inside ig_java). Of course, I'll still need a copy of core.jar within the ig_java directory as well.

Here's Kite.java:

```
package com.iragreenberg;

import processing.core.*;
import java.awt.*;

public class Kite extends PApplet {
```

```
// instance properties
private Point2D dummy;
private Vect2D dummyVel;
private int feetCount = 4;
private Point2D[] feet = new Point2D[feetCount];
private Vect2D[] feetDelta = new Vect2D[feetCount];
private Vect2D[] feetVel = new Vect2D[feetCount];
private Vect2D[] feetAccel = new Vect2D[feetCount];
private float botRadius = 45.0f;
private float[] springSpeed = new float[feetCount];
private float springSpdMin = .05f, springSpdMax = .1f;
private float damping = .9f;
private Point2D lead;
private Vect2D leadVel;
private int w, h;

// constructor
public Kite(int w, int h){
  this.w = w;
  this.h = h;
}

public void setup(){
   /* account for height of frame title bar,
   when sizing applet. getBounds() returns a Java
   Rectangle object that has 4 public properties:
   (x, y, width and height) */
  int titleBarHt = getBounds().y;
  size(w, h-titleBarHt);
  smooth();
  frameRate(30);
  dummy = new Point2D(width/2, height/2);
  dummyVel = new Vect2D(3.65f, 2.0f);
  lead = new Point2D(width/2, height/2);
  leadVel = new Vect2D(.04f, .05f);

  // set legs with some trig
  float theta = 0;
  float rad = 100;
  for (int i=0; i<feet.length; i++){
    feet[i] = new Point2D(dummy.x + cos(theta)*botRadius, ➥
            dummy.y + sin(theta)*botRadius);
    feetDelta[i] = new Vect2D(feet[i].x-dummy.x, feet[i].y-dummy.y);
    feetVel[i] = new Vect2D(0, 0);
    feetAccel[i] = new Vect2D(0, 0);
    theta += TWO_PI/feet.length;
    springSpeed[i] = random(springSpdMin, springSpdMax);
  }
}
```

```
public void draw(){
  background(255);
  drawBot();
  moveBot();
}

public void drawBot(){
  rectMode(CENTER);
  fill(0);
  for (int i=0; i<feet.length; i++){
    line(dummy.x, dummy.y, feet[i].x, feet[i].y);
    if (i<feet.length-1){
      line(feet[i].x, feet[i].y, feet[i+1].x, feet[i+1].y);
    }
    else {
      line(feet[i].x, feet[i].y, feet[0].x, feet[0].y);
    }
  }
  line(mouseX, mouseY, dummy.x, dummy.y);
  fill(255);
}

public void moveBot(){
  Vect2D[] feetMotionDelta = new Vect2D[feetCount];

  for (int i=0; i<feet.length; i++){
    feetMotionDelta[i] = new Vect2D((dummy.x-feet[i].x+ ➥
           feetDelta[i].vx)*springSpeed[i],
    (dummy.y-feet[i].y+feetDelta[i].vy)*springSpeed[i]);

    feetAccel[i].vx+= feetMotionDelta[i].vx;
    feetAccel[i].vy+= feetMotionDelta[i].vy;

    feetVel[i].vx += feetAccel[i].vx;
    feetVel[i].vy += feetAccel[i].vy;

    feet[i].x = feetVel[i].vx ;
    feet[i].y = feetVel[i].vy ;

    feetAccel[i].vx *= damping;
    feetAccel[i].vy *= damping;

    float rotAng = atan2(mouseY-(height+100), mouseX-width/2);
    dummy.x = mouseX + cos(rotAng)*200;
    dummy.y = mouseY + sin(rotAng)*200;
  }
}

/* two simple inner classes to help keep
  things semantically clear */
  class Point2D{
    float x, y;

    Point2D(float x, float y){
      this.x = x;
      this.y = y;
    }
  }

  class Vect2D{
    float vx, vy;

    Vect2D(float vx, float vy){
      this.vx = vx;
      this.vy = vy;
    }
  }
}
```

This class doesn't include any new concepts, so I'll leave it to you to sort out the Processing code. The simple springing physics in the class is based on the springing example I included in Chapter 11 (inspired by Keith Peters).

Next is the KiteController.java class:

```
import com.iragreenberg.Kite;
import java.awt.*;
import java.awt.event.*;
import java.applet.Applet;

public class KiteController extends Frame {

  private int w = 400, h = 422;

  // constructor
  public KiteController() {
    // call to superclass needs to come first in constructor
    super("Go Fly a Kite");

    // set up frame (which will hold applet)
    setSize(w, h);
    setLayout(new FlowLayout(FlowLayout.LEFT, 0, 0));

    // Instantiate Applet object
    Applet kite = new Kite(w, h);

    // add Applet component to frame
    add(kite);

    // won't allow frame to be resized
    setResizable(false);
    // allow window and application to be closed
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });

    // Next comment taken directly from PApplet class:
    /* "...ensures that the animation thread is started
     and that other internal variables are properly set."*/
    kite.init();
  }

  public static void main(String[] s){
    new KiteController().setVisible(true);
  }
}
```

You'll compile this program following exactly the same structure you did with the last program. The compile command is as follows:

```
javac -classpath .:core.jar KiteController.java
```

And the Java command to run it also follows the same structure:

```
java -classpath .:core.jar KiteController
```

Figure C-6 shows the initial output of the Kite application in a single window.

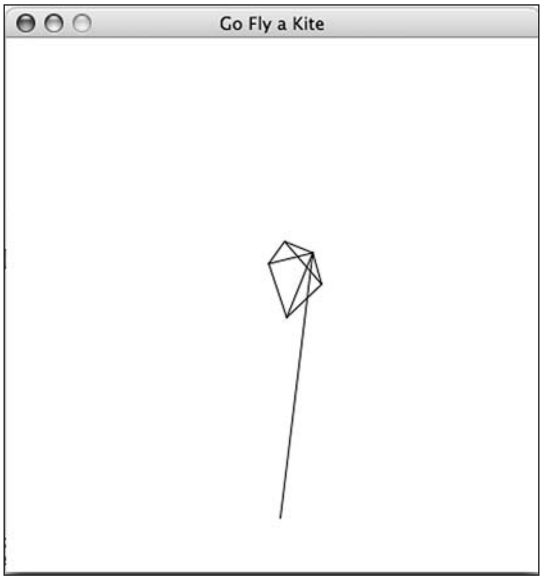


Figure C-6. Initial Kite application in a single window

Building an AWT-based GUI

Next, I'll add a button to the bottom of the application that will increment the number of sides to the kite each click. Here's the revised KiteController.java code, with the changes displayed in bold:

```
import com.iragreenberg.Kite;
import java.awt.*;
import java.awt.event.*;
// import java.applet.Applet;

public class KiteController extends Frame {

  private Button b;
  private int buttonH = 50;
  private int w = 400, h = 422;
  private Kite kite;


  // constructor
  public KiteController() {
    // call to superclass needs to come first in constructor
    super("Go Fly a Kite");

    // set up frame (which will hold applet)
    setSize(w, h);
    setLayout(new BorderLayout());
    setBackground(new Color(100, 100, 100));

    // create Button
    b = new Button("Increase Kite Sides");

    // Instantiate Kite object
    kite = new Kite(w, h-buttonH);

    // add Kite object component to frame
    add(kite, BorderLayout.CENTER);

    // add Button
    b.setBounds(0, 0, w, buttonH);
    add(b, BorderLayout.SOUTH);

    // won't allow frame to be resized
    setResizable(false);

    // allow window and application to be closed
    addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        System.exit(0);
      }
    });

    b.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent e) {
        kite.setFeetCount(1);
      }
    });

    // Next comment taken directly from PApplet class:
    /* "...ensures that the animation thread is started
     and that other internal variables are properly set."*/
    kite.init();
  }

  public static void main(String[] s){
    new KiteController().setVisible(true);
  }
}
```

Next, I'll update the Kite.java class, which only gets one new addition, a setFeetCount(int val) method. Add the following method to the Kite.java class. The method can be put anywhere within the class, but (of course) it shouldn't be put within another method.

```
// add to Kite.java
// dynamically change kite parameters
public void setFeetCount(int val){
    // reinitialize arrays
   feetCount += val;
   feet = new Point2D[feetCount];
   feetDelta = new Vect2D[feetCount];
   feetVel = new Vect2D[feetCount];
   feetAccel = new Vect2D[feetCount];
   springSpeed = new float[feetCount];

  // reset values
  float theta = 0;
  float rad = 100;
  for (int i=0; i<feet.length; i++){
   feet[i] = new Point2D(dummy.x + cos(theta)*botRadius, ➥
            dummy.y + sin(theta)*botRadius);
   feetDelta[i] = new Vect2D(feet[i].x-dummy.x, feet[i].y-dummy.y);
   feetVel[i] = new Vect2D(0, 0);
   feetAccel[i] = new Vect2D(0, 0);
   theta += TWO_PI/feet.length;
   springSpeed[i] = random(springSpdMin, springSpdMax);
  }
 }
```

I'll leave it to you to compile and run this example. (Hint: Just use the same two commands you used for the previous example.) The output is shown in Figure C-7. I'll also leave it to you to decipher the new AWT code I added; it's actually pretty straightforward. If you get stuck, I recommend doing a little API diving at `http://java.sun.com/j2se/1.4.2/docs/api/`.

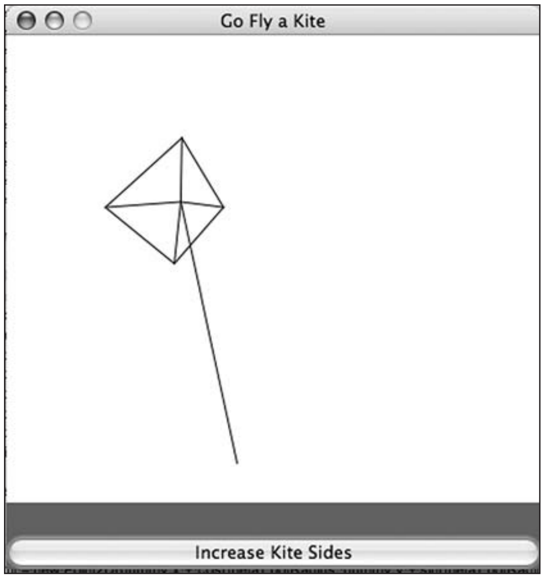


Figure C-7. AWT application

Building a Swing-based GUI

Finally, I'll leave you with a Swing-based GUI example. Following are the updated `KiteController.java` and `Kite.java` classes. There were lots of changes, so I include the full code. I've added comments throughout and tagged the new stuff I added. Enjoy, and happy creative coding.

```
// updated KiteController.java class
import com.iragreenberg.Kite;
import java.awt.*;
//import java.awt.event.*; - don't need anymore

// new
import javax.swing.event.*;
import javax.swing.*;

public class KiteController extends JPanel {

  private int w = 600, h = 422;
  private Kite kite;
```

```
// constructor
public KiteController() {

  setBackground(new Color(180, 180, 220));

  /* create 2 pane across layout
   left pane holds processing PApplet
   right pane holds user widgets */
  setLayout(new BoxLayout(this, BoxLayout.LINE_AXIS));

  // Instantiate Kite object
  kite = new Kite(400, 400);

  // right tools pane
  JPanel rtPanel = new JPanel();
  rtPanel.setOpaque(false);
  rtPanel.setLayout(new BoxLayout(rtPanel, ➥
           BoxLayout.PAGE_AXIS));

  JLabel jl = new JLabel("   Drag the Kite String");

  // kite radius slider
  JSlider js0 = new JSlider(JSlider.HORIZONTAL, 10, 100, 40);
  js0.setBorder(BorderFactory.createTitledBorder("Kite Radius"));
  js0.setMajorTickSpacing(30);
  js0.setMinorTickSpacing(4);
  js0.setPaintTicks(true);
  js0.setPaintLabels(true);

  // handle js0 change events
  js0.addChangeListener(new ChangeListener() {
    public void stateChanged(ChangeEvent evt) {
      JSlider slider = (JSlider)evt.getSource();

      if (!slider.getValueIsAdjusting()) {
        kite.setRadius(slider.getValue());
      }
    }
  }
  );

  // kite sides slider
  JSlider js1 = new JSlider(JSlider.HORIZONTAL, 3, 63, 4);
  js1.setBorder(BorderFactory.createTitledBorder("Kite Sides"));
  js1.setMajorTickSpacing(12);
  js1.setMinorTickSpacing(1);
  js1.setPaintTicks(true);
  js1.setPaintLabels(true);
```

```
// handle js1 change events
js1.addChangeListener(new ChangeListener() {
  public void stateChanged(ChangeEvent evt) {
    JSlider slider = (JSlider)evt.getSource();

    if (!slider.getValueIsAdjusting()) {
      kite.setFeetCount(slider.getValue());
    }
  }
}
);

// spring speed slider
JSlider js2 = new JSlider(JSlider.HORIZONTAL, 1, 10, 5);
js2.setBorder(BorderFactory.createTitledBorder("Spring Speed"));
js2.setMajorTickSpacing(3);
js2.setMinorTickSpacing(1);
js2.setPaintTicks(true);
js2.setPaintLabels(true);

// handle js2 change events
js2.addChangeListener(new ChangeListener() {
  public void stateChanged(ChangeEvent evt) {
    JSlider slider = (JSlider)evt.getSource();

    if (!slider.getValueIsAdjusting()) {
      kite.setSpringSpeed(slider.getValue()/20.0f);
    }
  }
}
);


// spring damping slider
JSlider js3 = new JSlider(JSlider.HORIZONTAL, 1, 10, 4);
js3.setBorder(BorderFactory.createTitledBorder("Spring Damping"));
js3.setMajorTickSpacing(1);
js3.setMinorTickSpacing(1);
js3.setPaintTicks(true);
js3.setPaintLabels(true);

// handle js3 change events
js3.addChangeListener(new ChangeListener() {
  public void stateChanged(ChangeEvent evt) {
    JSlider slider = (JSlider)evt.getSource();

    if (!slider.getValueIsAdjusting()) {
      kite.setSpringDamping(1.0f - slider.getValue()/30.0f);
    }
```

```
      }
    }
    );

    // assemble tools pane
    rtPanel.add(jl);
    rtPanel.add(Box.createRigidArea(new Dimension(0,10)));
    rtPanel.add(js0);
    rtPanel.add(Box.createRigidArea(new Dimension(0,8)));
    rtPanel.add(js1);
    rtPanel.add(Box.createRigidArea(new Dimension(0,8)));
    rtPanel.add(js2);
    rtPanel.add(Box.createRigidArea(new Dimension(0,8)));
    rtPanel.add(js3);


    // Next comment taken directly from PApplet class:
    /* "...ensures that the animation thread is started
     and that other internal variables are properly set."*/
    kite.init();

    // add panes to larger JPanel
    add(kite);
    // create some space between panes
    add(Box.createRigidArea(new Dimension(4,0)));
    add(rtPanel);
    // add a little margin on right of panel
    add(Box.createRigidArea(new Dimension(4,0)));
  }

  // create external JFrame
  private static void createGUI() {
    // create new JFrame
    JFrame jf = new JFrame("Kite Swing Application");

    // this line allows program to exit
    jf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    // You add things to the contentPane in a JFrame
    jf.getContentPane().add(new KiteController());

    // keep window from being resized
    jf.setResizable(false);

    // size frame
    jf.pack();

    // make frame visible
    jf.setVisible(true);
  }
  public static void main(String[] args) {
    // Recommended way to create a Swing GUI
    javax.swing.SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        createGUI();
      }
    }
    );
  }
}
```

Following is the updated Kite.java class. A screenshot of the final Swing GUI is shown in Figure C-8.

```
package com.iragreenberg;

import processing.core.*;
import java.awt.*;

public class Kite extends PApplet {

  // instance properties
  private Point2D dummy;
  private Vect2D dummyVel;
  private int feetCount = 4;
  private Point2D[] feet = new Point2D[feetCount];
  private Vect2D[] feetDelta = new Vect2D[feetCount];
  private Vect2D[] feetVel = new Vect2D[feetCount];
  private Vect2D[] feetAccel = new Vect2D[feetCount];
  private float botRadius = 45.0f;
  private float[] springSpeed = new float[feetCount];
  private float springSpdMin = .05f, springSpdMax = .1f;
  private float damping = .9f;
  private Point2D lead;
  private Vect2D leadVel;
  private int w, h;

  //NEW
  private int stringBaseX;
  private int stringBaseY;
  private float theta = 0;


  // constructor
  public Kite(int w, int h){
    this.w = w;
    this.h = h;
  }
```

```
public void setup(){
  /* account for height of frame title bar,
   when sizing applet. getBounds() returns a Java
   Rectangle object that has 4 public properties:
   (x, y, width and height) */
  int titleBarHt = getBounds().y;
  size(w, h-titleBarHt);
  smooth();
  frameRate(30);

  // NEW
  dummy = new Point2D(width/2, height/4);

  dummyVel = new Vect2D(3.65f, 2.0f);
  lead = new Point2D(width/2, height/2);
  leadVel = new Vect2D(.04f, .05f);

  // NEW
  stringBaseX = width/2;
  stringBaseY = height;

  // set legs with some trig
  for (int i=0; i<feet.length; i++){
    feet[i] = new Point2D(dummy.x + cos(theta)*botRadius, ➥
           dummy.y + sin(theta)*botRadius);
    feetDelta[i] = new Vect2D(feet[i].x-dummy.x, feet[i].y-dummy.y);
    feetVel[i] = new Vect2D(0, 0);
    feetAccel[i] = new Vect2D(0, 0);
    theta += TWO_PI/feet.length;
    springSpeed[i] = random(springSpdMin, springSpdMax);
  }
}

public void draw(){
  background(255);
  drawBot();
  moveBot();
}

public void drawBot(){
  rectMode(CENTER);
  fill(0);
  for (int i=0; i<feet.length; i++){
    line(dummy.x, dummy.y, feet[i].x, feet[i].y);
    if (i<feet.length-1){
      line(feet[i].x, feet[i].y, feet[i+1].x, feet[i+1].y);
    }
    else {
      line(feet[i].x, feet[i].y, feet[0].x, feet[0].y);
    }
  }
  // NEW
  line(stringBaseX, stringBaseY, dummy.x, dummy.y);
  if (mousePressed){
    stringBaseX = mouseX;
    stringBaseY = mouseY;
  }
  fill(255);
}

public void moveBot(){
  Vect2D[] feetMotionDelta = new Vect2D[feetCount];

  for (int i=0; i<feet.length; i++){
    feetMotionDelta[i] = new Vect2D((dummy.x-feet[i].x+ ➥
           feetDelta[i].vx)*springSpeed[i],
    (dummy.y-feet[i].y+feetDelta[i].vy)*springSpeed[i]);

    feetAccel[i].vx+= feetMotionDelta[i].vx;
    feetAccel[i].vy+= feetMotionDelta[i].vy;

    feetVel[i].vx += feetAccel[i].vx;
    feetVel[i].vy += feetAccel[i].vy;

    feet[i].x = feetVel[i].vx ;
    feet[i].y = feetVel[i].vy ;

    feetAccel[i].vx *= damping;
    feetAccel[i].vy *= damping;

    float rotAng = atan2(mouseY-(height+100), mouseX-width/2);
    //NEW
    if (mousePressed){
      dummy.x = mouseX + cos(rotAng)*200;
      dummy.y = mouseY + sin(rotAng)*200;
    }
  }
}

// dynamically change kite parameters
public void setFeetCount(int val){
  // reinitialize arrays
  feetCount = val;
  feet = new Point2D[feetCount];
  feetDelta = new Vect2D[feetCount];
  feetVel = new Vect2D[feetCount];
  feetAccel = new Vect2D[feetCount];
  springSpeed = new float[feetCount];
```

C

```
  // reset values
  float theta = 0;
  for (int i=0; i<feet.length; i++){
    feet[i] = new Point2D(dummy.x + cos(theta)*botRadius, ➥
           dummy.y + sin(theta)*botRadius);
    feetDelta[i] = new Vect2D(feet[i].x-dummy.x, feet[i].y-dummy.y);
    feetVel[i] = new Vect2D(0, 0);
    feetAccel[i] = new Vect2D(0, 0);
    theta += TWO_PI/feet.length;
    springSpeed[i] = random(springSpdMin, springSpdMax);
  }
}

// NEW
public void setRadius(int val){
  botRadius = val;
  for (int i=0; i<feet.length; i++){
    feet[i].x = dummy.x + cos(theta)*botRadius;
    feet[i].y = dummy.y + sin(theta)*botRadius;
    feetDelta[i].vx = feet[i].x-dummy.x;
    feetDelta[i].vy = feet[i].y-dummy.y;
    theta += TWO_PI/feet.length;
  }
}

// NEW
public void setSpringSpeed(float val){
  for (int i=0; i<feet.length; i++){
    springSpeed[i] = random(springSpdMin, val);
  }
}

// NEW
public void setSpringDamping(float val){
  damping = val;
}


/* two simple inner classes to help keep
 things semantically clear */
class Point2D{
  float x, y;

  Point2D(float x, float y){
    this.x = x;
    this.y = y;
  }
}
```

```
    class Vect2D{
      float vx, vy;

      Vect2D(float vx, float vy){
        this.vx = vx;
        this.vy = vy;
      }
    }
  }
```

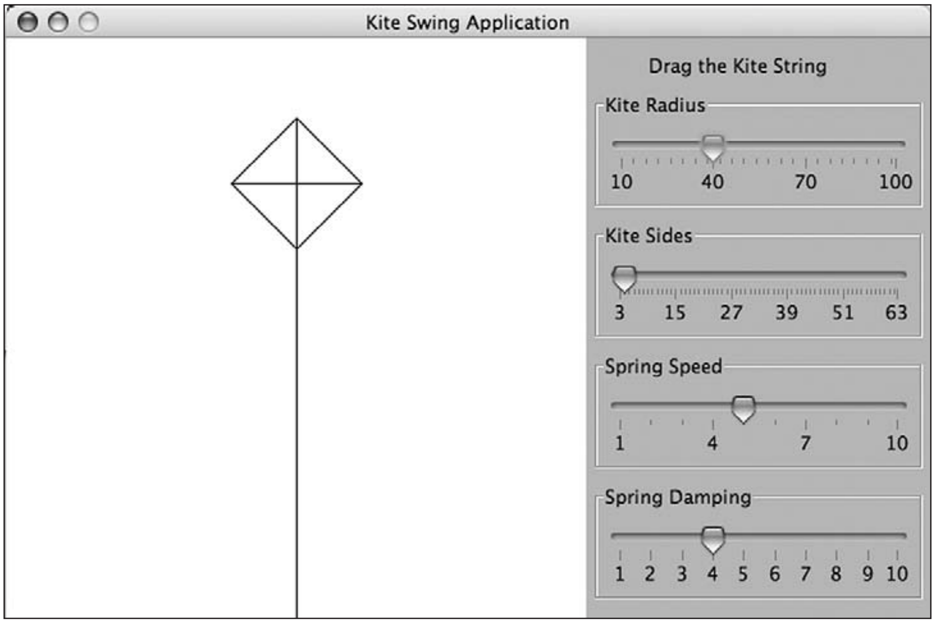


Figure C-8. Swing application